Virtual Dispatch Alternatives to Vtables

25 September, 2001

George Bosworth
CLR Software Architect

Agenda

- Observations and Biases
- Current Approach (V1) - vtables
- Alternative - stubs and heuristics
- Implementation
- Performance Measurements
- Observations

Observations and Biases

- Delayed binding is in general a good thing
  - loose coupling, resilience, reuse, ...
- Vtables are a bad thing in the large
  - space inefficient, tight bindings, ...
- Heuristics can mask a lot of ignorance and provide for the unexpected

Agenda

- Observations and Biases
- Current Approach (V1) - vtables
- Alternative - stubs and heuristics
- Implementation
- Performance Measurements
- Observations

Current Approach (V1)

- Method table contain
  - vtables for virtual functions
  - vtables for interface functions
- Interface Table maps interfaces for a type to the interface vtable in a method table

Method table

```
interface I1 {m1(); }
interface I2 {m2(); m3(); }
class A : I1 {m1(){..}; m2(){..}; }
```

A.m1, I1.m1
A.m2

```
class B : A, I2 {m3(){..}; }
```

A.m1, I1.m1
A.m2, I2.m2
B.m3, I2,m3

```
class C : B { I2.m2() {..}; }
```

A.m1, I1.m1
A.m2
B.m3
I2.m2
I2.m3

Method table

```
interface I1 {m1(); }
interface I2 {m2(); m3(); }
class A : I1 {m1(){..}; m2(){..}; }
```

A.m1, I1.m1
A.m2

```
class B : A, I2 {m3(){..}; }
```

A.m1, I1.m1
A.m2, I2.m2
B.m3, I2,m3

```
class C : B { I2.m2() {..}; }
```

A.m1, I1.m1
A.m2
B.m3
I2.m2
I2.m3

Calling Sequences

- virtual call: c.m2()

```
;virtual invocation of m2(), ecx = this ptr
mov  eax,[ecx]    ;get method table
call [eax+4]      ;dispatch thru virtual slot 1
```

- interface call: i2.m2()

```
;interface invocation of I2.m2(), ecx = this ptr
mov  eax,[ecx]    ;get method table
mov  eax,[eax+xx] ;get the interface map for class C
mov  eax,[eax+yy]  ;get the dispatch vtable of I2 in C
                   ;  which is virtual slot 3 in C
call [eax+0]       ;dispatch thru slot 0 of I2 in C
```

Interface Map

- Tightly packed two dimensional array
  - each class points to its row
  - interface Ids are monotonically increasing global index

Method Table Statistics

	Size in bytes						
program	# MT	MT	VT	Static	Field	GC	I/Fmap
helloworld.exe	269	53076	21788	14352	636	1688	2776
vswinformsdesigner.	2072	464008	236440	80860	8716	21144	25680
vsstartuppropertygri	761	185328	91512	39688	3684	6072	10888
winformsstarupcomp	705	159728	72340	39008	3604	5748	8008
xslttest.exe	647	115400	48612	25260	2688	5228	5144
oledbtest.exe	987	182044	77616	40840	3764	7860	8536

Winforms Hierarchical Slice

	Size in bytes						
class	MT	VT	Static	Field	GC	I/Fmap	parent
Object	100	16	40	0	0	0	
MarshalByRefObject	124	28	40	0	12	0	System.Object[mscorlib.dl
Component	156	52	20	4	20	16	System.MarshalByRefObj
Control	3664	1268	1772	416	28	136	System.ComponentModel.
ScrollableControl	1616	1280	116	4	36	136	System.Windows.Forms.C
ContainerControl	1640	1316	88	4	44	144	System.Windows.Forms.S
Form	2584	1380	716	248	52	144	System.Windows.Forms.C
SelfHost	1940	1388	240	64	60	144	System.Windows.Forms.F

Type and Slot Usage

helloworld						
threshold	#vt	#used	#slots	#used	#vt-slots	#calls
1	164	8	2096	26	268	46
2	164	4	2096	11	146	31
4	164	1	2096	2	29	12
8	164	1	2096	1	29	8

- #vt=total number of method tables loaded.
- #used=number of method tables with at least one virtual slot of the table that is invoked thru at least threshold times.
- #slots=total number of virtual slots in all loaded method tables.
- #used=total number of virtual slots that were invoked thru at least threshold times. Note this is not the same as the number of times a method implementation was invoked. A method implementation that is inherited is in several method tables, and hence is invocable thru several slots.
- #vt-slots=number of slots in the method tables with at least one slot that is invoked thru at least threshold times. I.e. the sum of the size of the vtables that are used frequently (frequently meaning have a slot that is invoked thru at least threashold times)
- #calls=total calls thru slots that were invoked at least threshold times.

A bigger program

vswinformsdesigner

threshold	#vt	#used	#slots	#used#	vt-slots	#calls
1	1967	449	57519	3721	22663	2174385
2	1967	414	57519	2874	21664	2173538
4	1967	369	57519	2142	20409	2171891
8	1967	349	57519	1702	19695	2169680
16	1967	299	57519	1341	18672	2165703
32	1967	249	57519	1040	17196	2159071
64	1967	217	57519	832	15905	2149739
128	1967	180	57519	613	13652	2130147
256	1967	144	57519	455	8514	2101325
512	1967	108	57519	332	6331	2055404
1024	1967	72	57519	204	4362	1958830
2048	1967	44	57519	117	1331	1839083
4096	1967	36	57519	76	1193	1721232
8192	1967	22	57519	43	735	1530862
16384	1967	13	57519	26	465	1348424
32768	1967	11	57519	17	439	1134720

Observations

- Most invocations go thru few slots
- Not many types actually instantiated
- Something weird going on in Winforms
  - 11 slots account for ½ the calls
  - 4 of those slots are in HashTable

Other programs

datadontlost

threshold	#M	#used	#slots	#used	#M-slots	#calls
1	611	76	16183	330	3679	35716
2	611	62	16183	277	3077	35655
4	611	47	16183	217	2690	35613
8	611	40	16183	169	2333	35230
16	611	35	16183	119	2140	34700
32	611	30	16183	91	1805	34132
64	611	20	16183	54	1439	32282
128	611	13	16183	30	1016	29770
256	611	10	16183	21	668	28248
512	611	7	16183	17	410	26938
1024	611	5	16183	8	171	20988
2048	611	3	16183	4	109	14656
4096	611	1	16183	1	60	4320

objectmodeltest

threshold	#M	#used	#slots	#used	#M-slots	#calls
1	927	138	27481	689	6556	126117
2	927	124	27481	613	6024	126041
4	927	103	27481	494	5333	125786
8	927	97	27481	413	5089	125395
16	927	75	27481	346	3965	124666
32	927	59	27481	293	3346	123477
64	927	47	27481	214	2609	119894
128	927	39	27481	158	2399	114481
256	927	29	27481	101	1926	106700
512	927	23	27481	59	1677	09227
1024	927	19	27481	29	1324	67514
2048	927	10	27481	12	498	43212
4096	927	3	27481	5	109	26236

Limitations

- vtables require the code generator (jit) to bind to a slot number
  - cannot decide later (dynamic invocation for scripting)
  - use another binding approach (multimethods)
  - detect out of range (version resiliency, extensibility)
- Must be built whether needed or not

What is needed

- Mapping function from:
  - (calling frame, ref token)
- to:
  - method implementation address
- Vtables are a restricted form of this mapping function where:
  - calling frame is just the <this type>
  - ref token is (contract, member)
    - contract = virtual or interface id
    - member = vtable or interface table slot #

Asymmetries to leverage

- Few actual types instantiated
- Small number of slots used

- Assumption we will make:
  - most call sites are mostly monomorphic

Alternative – Stub Dispatch

- Type and method specific stubs
- Different kinds of stubs
  - mostly monomorphic
  - polymorphic
- Stubs built on demand as needed
- Heuristics used to choose and adjust the kind of stub in use

Simple Logical Stub Model

- generated call sites:

```
;aligning pad/nop if necessary (0-3 bytes)
call stubaddress
```

- initial stubaddress points to:

```
push #methRefToken ;identifies contract,member
jmp  resolver
```

- after use, stubaddress points to:

```
cmp [ecx],#expectedMT  ;check the <this> type
jne fail
jmp methodAddress      ;transfer to target impl
fail: jmp resolverStubForMethodRef
```

Trace based Measurements

- Traces from a modified fjit
- Stub model consumed traces
  - *call tick* is the running counter of the number of virtual and interface calls made since the start of the program.
  - *sites* is the number of call sites created since the prior *call tick*.
  - *stubs* is the number of new dispatch stubs made since the prior *call tick*.
  - *misses* is the number of dispatch stub misses/failures, since the prior *call tick*, i.e. the number of times the actual runtime type of the *this* argument did not match the expected value in some dispatch stub.
  - *jits* is the number of methods jitted since the prior *call tick*.
  - *classes* is the number of newly encountered classes since the prior *call tick*.

Trace for OLEDBTEST

call tick	sites	stubs	misses	jits	classes
1	2	1	1	82	18
2	1	1	1	3	0
3	2	1	1	2	0
4	2	1	1	2	0
5	3	1	1	7	2
6	1	0	0	1	0
7	0	1	1	0	0
8	0	1	1	1	0
9	1	0	0	2	0
10	0	0	0	0	0
11	1	1	1	5	0
20	40	2	5	19	4
30	0	0	4	0	0
40	0	0	3	0	0
50	0	0	3	0	0
60	0	0	4	0	0
70	0	0	3	0	0
80	0	0	3	0	0
90	0	0	4	0	0

(Cont) Trace for OLEDBTEST

call tick	sites	stubs	misses	jits	classes
1000	0	0	0	0	0
2000	8	2	9	5	1
3000	0	0	0	0	0
4000	0	0	0	0	0
5000	55	18	39	54	6
6000	134	38	108	171	21
7000	0	0	0	2	0
8000	0	0	0	0	0
9000	0	0	0	0	0
10000	1	1	1	2	0
20000	0	0	0	0	0
30000	0	0	0	0	0
40000	0	0	0	0	0
50000	0	0	0	0	0
60000	0	0	0	0	0
70000	39	9	41	27	2
80000	14	4	28	40	5
90000	0	0	0	0	0
100000	0	0	0	0	0
200000	16	1	20	2	0

Summary for OLEDBTEST

jitted sites	3273	called sites	2130	
stub calls	864968	misses	24457	(2.8%)
stubs made	897	tokens	121	
stubs/site	1.17	stub refs	5.54	

- small number of stubs and tokens
- low miss rate
- program has:
  - 987 types
  - over 10,000 methods
  - 20,000 vtable slots

Mono or Polymorphic ?

miss%	calls	sites	misses	stubs
0-0:	758512	2023	108	665
1-1:	8761	4	121	12
2-2:	2714	4	61	10
3-3:	6277	6	221	19
4-4:	1973	4	91	13
6-6:	166	2	10	4
7-7:	2768	2	207	4
8-8:	9461	2	830	12
10-19:	36522	9	5765	37
20-29:	13949	14	3676	62
30-39:	2873	10	985	29
40-49:	3510	7	1543	37
50-59:	11472	27	6130	76
60-69:	1415	6	894	17
70-79:	4	1	3	2
80-89:	4507	7	3730	16
90-99:	84	2	82	12

Modified CLR built

- EE generates and manages stubs
- Jit generates stub calls for interface and virutal methods
- What wasn't done:
  - eliminate the vtables themselves
  - integrate stubs with app domain unloading
  - atomic update of call sites partially undone

Tokens

- tokens composed of two 32bit fields
  - contract
    - 0 means use the type of <this>
    - >0 means contract is the interface id
  - member
    - slot number in the vtable or interface table
- Since the numbers are small, it is usually (always) bit encoded in 31 bits

Call Sites

- with call site rewriting

```
mov eax,[ecx]     //preload the MT
call stub         //direct call to stub
```

- as pure code

```
mov eax,[ecx]     //preload the MT
call [stubcell]   //indirect call to stub
```

Lookup Stubs

- Call sites initially point to:

```
push token
jmp  ResolveWorkerStub
```

- ResolveWorkerStub enters the EE and causes a Dispatcher stub to be found (or created) that is specific to the type of the <this> pointer of the call
  - betting monomorphic until shown otherwise

Dispatcher Stubs

```
cmp eax,expectedMT
jne  failstub       ;must be fwd branch
jmp targetaddress
```

- Checks that the actual type of <this> is what is expected, and if so transfers to the method implementation
- fail stub will cause a lookup to be done and then pass control to the correct method.
- One per tuple (expectedMT, token)

Fail Stub

```
sub  [counter],1
jl   patchstub   ;must be fwd jmp
;fall into resolve stub
```

- Fail stub checks to see if we are failing a lot, counter is periodically inc'd
- Patch stub will patch the call site to point to the resolve stub
- Resolve stub will do a cache lookup
- One per token, i.e. dispatch stub share

Patch Stub

```
call PatchWorkerStub
jmp resolvestub
```

- PatchWorkerStub goes into the EE and decides whether or not the call site should be patched to point to the resolvestub directly, i.e. make it a polymorphic call site.

Resolve Stub

```
    push  edx
    shld  edx,eax,16
    lea   eax,[eax+edx+hashed_token]
    mov   edx,[ecx]
    and   eax,cache_mask
    mov   eax,[eax+cache]
    cmp   edx,[eax+mt]
    jne   miss              ;must be fwd jump
    cmp   [eax+token],token
    jne   miss              ;must be fwd jmp
    pop   edx
    jmp   [eax+target]      ;we hit
miss: pop edx
    push token
    jmp   ResolveWorkerStub
```

Resolve Stub

```
    push  edx
    shld  edx,eax,16
    lea   eax,[eax+edx+hashed_token]
    mov   edx,[ecx]
    and   eax,cache_mask
    mov   eax,[eax+cache]
    cmp   edx,[eax+mt]
    jne   miss              ;must be fwd jump
    cmp   [eax+token],token
    jne   miss              ;must be fwd jmp
    pop   edx
    jmp   [eax+target]      ;we hit
miss: pop edx
    push token
    jmp   ResolveWorkerStub
```

Resolve Stub (Cont)

- Resolve stubs to a cache lookup keyed by the type of <this> and the token.
- Cache contains the method impl address.
- Key to the cache hit rate is the hash fn used
  - current has not been extensively tuned

Agenda

- Observations and Biases
- Current Approach (V1) - vtables
- Alternative - stubs and heuristics
- Implementation
- Performance Measurements
- Observations

Simple calls - best case

25 inlined calls embedded in a loop
Loop is just the loop

	loop count	current jit	d-stub	r-stub
Loop	100000	0	0	0
Loop	1000000	0	0	0
Loop	10000000	47	47	31
Call	100000	15	15	15
Call	1000000	172	172	172
Call	10000000	1688	1672	1657
Static	100000	15	16	15
Static	1000000	172	172	188
Static	10000000	1672	1672	1656
Interface	100000	16	16	31
Interface	1000000	187	172	375
Interface	10000000	1906	1828	3469
Virtual	100000	16	32	31
Virtual	1000000	172	172	360
Virtual	10000000	1672	1828	3485

Richards – worse case

- Highly polymorphic
  - dna - straight port to c#
  - dav - everything is a virtual call
  - dai - everything is an interface call

	richards_dna	richards_dav	richards_dai
call sites	201	307	313
virtual calls	6579070	162065580	162065568
lookup stubs	46	48	89
dispatcher stubs	75	196	206
dispatcher calls	232	56615497	56615493
dispatcher misses	104	3417	3415
resolver stubs	34	40	77
resolver calls	6578942	105453500	105453490
resolver misses	9	298	105
call site writes	78	217	220
mono sites	76	153	156
poly sites	1	32	32
execution time	1078	5937	3219

The bad news

- The timings are very unstable. Small changes in unrelated portions of the EE can cause big jumps (factor of 2) in the above numbers
- Tried, unsuccessfully, to isolate via vtune.
- Built a vtune post processor that computed aggregate stats for the stubs themselves, still didn't isolate

Controlled Sequences

- Built a hardcoded set of code sequences that did the current virtual dispatch and the stub dispatch
- Varied the calling and the called environment.
- Basically a loop of 20,000 around 50 inlined call sites

Controlled Sequences (Cont)

- NOP(s) of 1 or 2 bytes are inserted between every 10 calls
- (CONTEXT_0) = no inter call instr overhead
- (CONTEXT_5) = 5 instr (dummy bp frame-like) inter call overhead
- Called routine was just a return, and was either Frameless or EBP Framed

Controlled Sequences (Cont)

	FRAMELESS		EBP FRAME	
	C_0	C_5	C_0	C_5
SMALL Context only	0.061	4.478	0.061	4.59
SMALL DIRECT	4.411	3.378	6.913	8.277
SMALL VIRTUAL	6.004	4.704	5.997	4.606
SMALL DIRECT_STUB	7.684	7.553	5.103	3.759
SMALL INDIRECT_STUB	9.534	9.103	6.999	4.917

- anomalies make one read the intel perf manuals really close

does it vary with code size?

	FRAMELESS		EBP FRAME	
	C_0	C_5	C_0	C_5
SMALL Context only	0.061	4.478	0.061	4.59
LARGE Context only	0.053	4.663	0.053	4.626
SMALL DIRECT	4.411	3.378	6.913	8.277
LARGE DIRECT	9.328	6.694	10.186	9.144
SMALL VIRTUAL	6.004	4.704	5.997	4.606
LARGE VIRTUAL	20.584	18.465	20.494	17.854
SMALL DIRECT_STUB	7.684	7.553	5.103	3.759
LARGE DIRECT_STUB	11.269	10.925	9.369	8.463
SMALL INDIRECT_STUB	9.534	9.103	6.999	4.917
LARGE INDIRECT_STUB	19.657	18.519	17.803	15.659

- small - loop 20,000 over 50 inlined sites
- large - loop 1,000 over 1,000 inlined sites

Say hello to the BTB

- Branch Transfer Buffer on the Pentium
  - all my runs were on a 933 Mhz PIII with 800Mhz RDRAM

Branch basics on pentiums

- The BTB is check on every instr fetch of any kind. If it hits, it turns all transfers into zero or near zero cost
- If no BTB hit, fall thru is free
- If non- fall thru but staticly predicted, e.g. a jmp relative, cost is a half dozen or so cylces
- If not statically predicted, e.g an indirected call or a mis-prediction cost is a score or more cycles -and this doesn't include any cache or memory fetches at all.
- BTB is 512 cells or so, i.e. small.

Simple Calls - revisited

25 inlined calls embedded in a loop Loop is just the loop					2500 inlined calls embedded in a loop			
	loop count	current jit	d-stub	r-stub	loop count	current jit	d-stub	r-stub
Loop	100000	0	0	0	1000	0	0	0
Loop	1000000	0	0	0	10000	0	0	0
Loop	10000000	47	47	31	100000	0	0	0
Call	100000	15	15	15	1000	47	47	47
Call	1000000	172	172	172	10000	500	484	500
Call	10000000	1688	1672	1657	100000	5032	5000	5000
Static	100000	15	16	15	1000	47	47	62
Static	1000000	172	172	188	10000	500	484	485
Static	10000000	1672	1672	1656	100000	5000	4984	5000
Interface	100000	16	16	31	1000	63	31	62
Interface	1000000	187	172	375	10000	718	344	485
Interface	10000000	1906	1828	3469	100000	7204	3578	5047
Virtual	100000	16	32	31	1000	62	31	46
Virtual	1000000	172	172	360	10000	563	343	500
Virtual	10000000	1672	1828	3485	100000	5578	3579	5000

- indirections, like vtable dispatch, fall apart in the large.

Agenda

- Observations and Biases
- Current Approach (V1) - vtables
- Alternative - stubs and heuristics
- Implementation
- Performance Measurements
- Observations

Conclusion

- Stubs can be performat.
- Perf is hard to measure
- Things like BTB make tuning almost impossible
- Transfers of control carry a large hidden perf cost on Pentiums, since any transfer will put pressure on the BTB
  - all jmps, jccs, calls, rets of any kind

Continuations, anyone?

Nick Benton
MSR Cambridge

What's a continuation?

- The "rest of the computation"
- Having first-class continuations in a language or VM means being able to *reify* the current "rest of the computation" into a value which can then be passed around and invoked later
  - Possibly never
  - Possibly several times
- To a first approximation, grab the stack and turn it into an object on the heap
- This single facility allows one to build lots of fancy control structures which would normally be implemented separately

For example

- Exceptions
- Backtracking a la Prolog
- Coroutines
- Lightweight threads (CML)
- Stateless servers
- Mobile computations (migratory apps)
- Security features
- Time-travel debugging
- Arbitrary monads – adding step-counting, instrumentation, non-determinism, probabilistic computation....

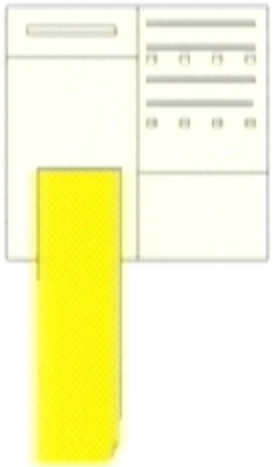
Monadic reflection using call/cc

- Add nondeterminism (for example) to already-compiled code

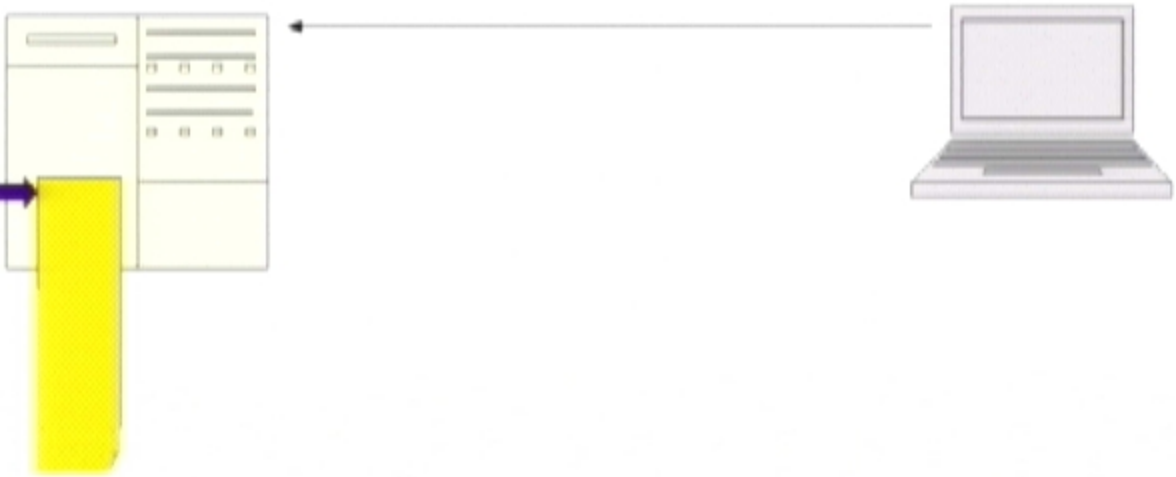
```
val pick : 'a list -> 'a
val fail : unit -> 'a
val results : (unit->'a)->'a list

- results (fn () =>
  let val x = pick [1,2]
      val y = pick [2,5]
  in if x = y then fail() else x+y
  end)
> [3,6,7] : int list
```

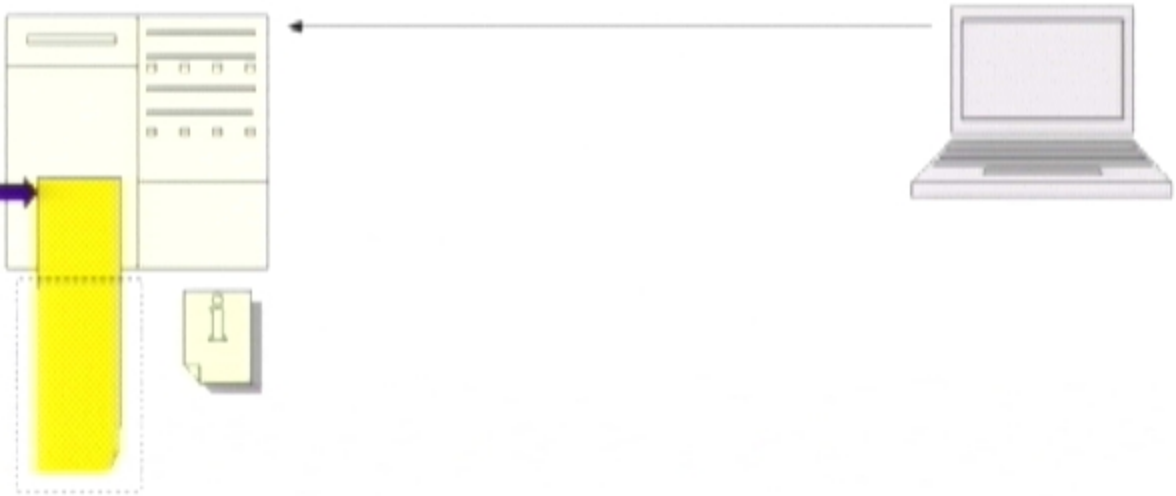
e.g. stateless servers



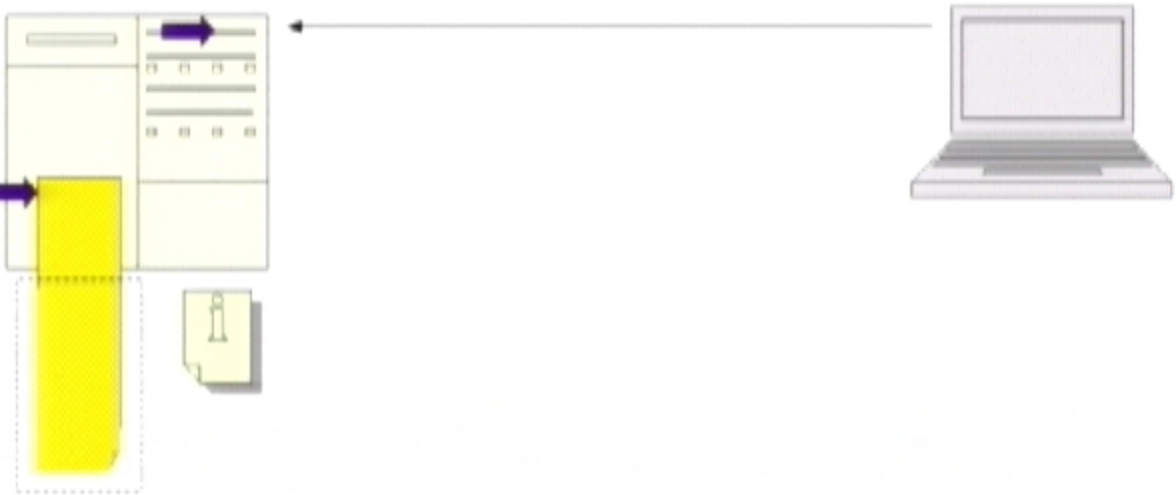
e.g. stateless servers



e.g. stateless servers



e.g. stateless servers



History

- First invented in denotational semantics in order to give a mathematical account of the meaning of existing control operators, such as jumps
- Then added to programming languages such as Scheme and SML/NJ (call/cc)
- Still interesting research going on
  - More refined control operators (prompt/reset, hierarchies,...)
  - Connection with classical logic

Continuation Passing Style

- There's a uniform translation of CBV (or, indeed, CBN) into CPS
  - Every function gets an extra argument "what to do with the result"
  - Every call becomes a tail call

Why should a VM support first-class continuations?

- Some languages have them and compiling via explicit CPS to the runtime gives poor performance (especially without tail calls that work...)
- Similarly, some other languages have features (e.g. backtracking, large numbers of very lightweight threads) that can be implemented more efficiently with continuations
- It's potentially simpler than adding lots of control-related stuff in an ad hoc way
- A general purpose runtime should provide general purpose mechanisms

Why should a VM support first-class continuations?

- Some languages have them and compiling via explicit CPS to the runtime gives poor performance (especially without tail calls that work...)
- Similarly, some other languages have features (e.g. backtracking, large numbers of very lightweight threads) that can be implemented more efficiently with continuations
- It's potentially simpler than adding lots of control-related stuff in an ad hoc way
- A general purpose runtime should provide general purpose mechanisms

Why *shouldn't* a VM support first class continuations?

- Scary
- Almost certainly interact badly with existing features (e.g. security)
- Rule out some optimisations which might mean paying an efficiency cost in the common case of languages which don't use them

Conclusion

- None, really
- But it's something that seems worth thinking about…

LZ77 data compression

- Before: Blah blah blah.
- After: Blah b<-5,8>.
- Encoding: 1111 1101 B l a h ' ' b 0x5008 '.'
- Recast as "instructions" to a decompressor:

```
Lit 'B'
Lit 'l'
Lit 'a'
Lit 'h'
Lit ' '
Lit 'b'
Echo 5,8
Lit '.'
```

The echo instruction

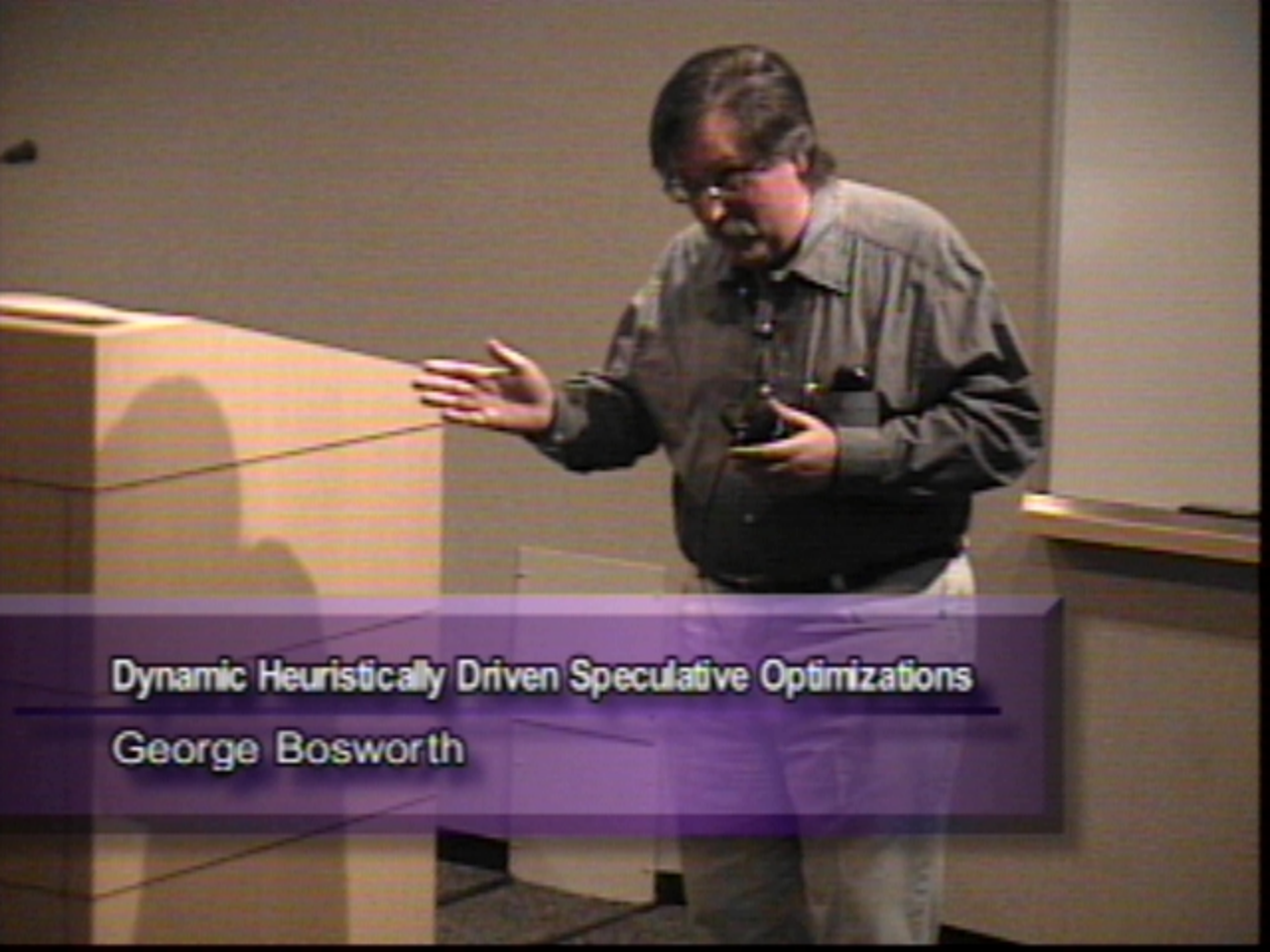
- Example:

```
ldc.i4 12
add
ldind.i4
...
echo 30,3
```

- Like `#include` but on-the-fly
- Like `JSR` but caller delimits entry/exit without overhead polluting the stream

Summary

- Cuts LCC byte-code by ~40% == 0.60x
- Echo suits interpreters:
  - Clients that can't JIT or even decompress
  - e.g., "feature" code in cell phones
- Related tools:
  - Automatic generation of compressed bytecodes
  - High-tech code and XML compression
    (esp. for thin or costly pipes, service packs, ROM)
  - Simple, table-driven code generators and JITs


Dynamic Heuristically Driven Speculative Optimizations
George Bosworth